CSE 425 Studio: Functional Programming II

These studio exercises are intended to give you additional experience with functional programming features available in C++, including function objects, delayed evaluation, and continuations.

In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises in an e-mail to the course account. Students who are more familiar with the material are encouraged to partner with those who are less familiar with it, and asking questions of your classmates and your professor during the studio sessions is highly encouraged as well.

Please record your answers as you work through the following exercises. After you have finished please email your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu course account, with subject line "Functional Programming Studio II". The enrichment exercise is optional but is a good way to dig into the material a little deeper, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people of your choice (preferably a mix of people familiar and unfamiliar with C++) and write down the names of the team members as the answer to this first exercise.

2. Create a new Visual Studio 2013 project for this studio (for example, named **`FunctionalStudioII`** or something similar that identifies which studio this is for). Modify the signature of the main function in your project's main source file (e.g., **`FunctionalStudioII.cpp`**) so that it matches the standard portable (e.g., between Windows and Linux) main function signature (and variable names) for a C++ program: **`int main (int argc, char * argv[])`**.

Outside of your main function, declare a function that takes an unsigned integer by value as its only parameter, and computes and returns the factorial of the passed value. This function should check whether the passed value is less than or equal to 1 and if so should simply return 1; otherwise the function should return the result of multiplying the passed value by the result of calling itself recursively with one less than the passed value.

In your main function, call that function with an unsigned integer value that is larger than 3, and print out (to the standard output stream) both the value that was passed into that call and the value returned by that call. Build and run your program, and as the answer to this exercise, please show both the code you wrote and the program's output.

3. Declare and define a class with a private const unsigned integer member variable and a public constructor that takes a single unsigned integer by value and uses it to initialize the member variable. Move the recursive function you implemented in exercise 2 into the class as a private member function. Also declare and define a public function call operator (**`operator()`**) that returns the result of calling that recursive private member function with the value of the member variable.

In your main function, construct an object of that class type using the same value with which you called the recursive (non-member) function in the previous exercise, and print out that value and the result of calling the object as though it were a function (i.e., invoking its function call operator). Build and run your program, and as the answer to this exercise show your code and the output the program produced.

4. Add two non-const private unsigned integer member variables to the class, one to store the current (intermediate or final) result of the factorial computation, and one to store the remaining value for which the factorial still needs to be computed.

Modify the function call operator so that it first sets the value of the current result member variable to 1 and then assigns the value of the const member variable from the previous exercise (i.e., the value that was provided to the constructor) to the member variable that keeps track of the remaining factorial that needs to be computed. The function call operator should then call the private recursive member function (with no arguments – see next) and then return the value that is stored in the current result member variable.

Modify the private recursive function so that instead of taking an input parameter by value, it takes no parameters, and has a void return type. It should check whether the value of the member variable that keeps track of the remaining factorial to be computed is less than or equal to 1 and if so should simply return; otherwise the function should (1) multiply the current result by the remaining factorial value, (2) decrement (by one) the value of the member variable that keeps track of the remaining factorial to compute, and (3) call itself recursively.

Build and run your program, and as the answer to this exercise show your code and the output the program produced.

5. Modify the private member function so that it no longer calls itself recursively (i.e., so that it now computes only a single step of the recursion if one is needed). Add a new public **`run`** member function that takes an unsigned integer by value and makes that many calls to the private member function you just modified.

Modify the function call operator so that instead of calling the private member function you just modified, it passes the value of the const member variable into a call to the **`run`** member function you just added. Build and run your program, and as the answer to this exercise show your code and the output the program produced.

6. Declare and define a new public (accessor) member function that returns the value of the current result member variable.

Modify the run method so that after it has finished calling the private member function that computes one step of the factorial computation as many times as it was told to do so, it returns a **`bool`** value indicating whether or not there is any remaining computation to be done (i.e., whether or not the member variable that tracks the remaining factorial value to be computed is greater than 1).

Modify the function call operator so that it only initializes the member variables, and does not call the run method or return a value.

Modify your main function so that after it calls the object's function call operator, it repeatedly (1) calls the run method with a given value for the number of computation steps steps to perform in that run, and (2) prints out the value returned by the accessor member function to show the result computed so far, until the run method returns false. Build and run your program, and as the answer to this exercise show your code and the output the program produced.

PART II: ENRICHMENT EXERCISE (optional, feel free to skip or to try variations that interest you)

7. Play around with different variations on the behavior of the member function that accesses the result, with respect to whether or not there is still more work to be done. For example, you could have the accessor member function throw an exception if it is invoked when there is still remaining work to be done. As the answer to this exercise, show your code and the output of one or more runs of your program, explain briefly what you tried, and describe what you observed when you did that.